DOCUMENT RESUME

ED 477 013 IR 021 741

AUTHOR	Garner, Stuart
TITLE	Reducing the Cognitive Load on Novice Programmers.
PUB DATE	2002-06-00
NOTE	7p.; In: ED-MEDIA 2002 World Conference on Educational Multimedia, Hypermedia & Telecommunications. Proceedings (14th, Denver, Colorado, June 24-29, 2002); see IR 021 687.
AVAILABLE FROM	Association for the Advancement of Computing in Education (AACE), P.O. Box 3728, Norfolk, VA 23514. Tel: 757-623-7588; e-mail: info@aace.org; Web site: http://www.aace.org/DL/.
PUB TYPE	Reports - Descriptive (141) -- Speeches/Meeting Papers (150)
EDRS PRICE	EDRS Price MF01/PC01 Plus Postage.
DESCRIPTORS	Academic Achievement; *Cognitive Processes; Computer Assisted Instruction; *Computer Software Development; *Instructional Design; *Instructional Materials; *Learning Theories; *Material Development; Programming
IDENTIFIERS	*Cognitive Load

ABSTRACT

Computer programming is a domain of knowledge that is generally considered difficult by students, many of whom experience low levels of achievement and become disillusioned. This paper suggests that cognitive load theory needs to be taken into account when designing instructional materials for this domain. The cognitive load that is experienced by a student can be considered to be made up of three types: intrinsic, extraneous, and germane. Computer programming has a high intrinsic load and it is, therefore, necessary to reduce the extraneous load as much as possible by using techniques such as the study of programming examples. Germane cognitive load can then be applied by removing certain parts of the solutions to the examples and then requiring students to complete these part-complete solutions, thereby encouraging schemata creation in long-term memory. A new software tool called CORT (Code Restructuring Tool) has been created which utilizes this part-completion method. (Contains 20 references and 2 figures.) (Author)

Reproductions supplied by EDRS are the best that can be made from the original document.

PERMISSION TO REPRODUCE AND DISSEMINATE THIS MATERIAL HAS BEEN GRANTED BY

G.H. Marks

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)
✓ This document has been reproduced as received from the person or organization originating it.
☐ Minor changes have been made to improve reproduction quality.
• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

Reducing the Cognitive Load on Novice Programmers

Stuart Garner
School of Management Information Systems
Edith Cowan University
Western Australia
s.garner@ecu.edu.au

Abstract: Computer programming is a domain of knowledge that is generally considered difficult by students, many of whom experience low levels of achievement and become disillusioned. This paper suggests that cognitive load theory needs to be taken into account when designing instructional materials for this domain. The cognitive load that is experienced by a student can be considered to be made up of three types: intrinsic, extraneous, and germane. Computer programming has a high intrinsic load and it is therefore necessary to reduce the extraneous load as much as possible by using techniques such as the study of programming examples. Germane cognitive load can then be applied by removing certain parts of the solutions to the examples and then requiring students to complete these part-complete solutions thereby encouraging schemata creation in longterm memory. A new software tool called CORT (Code Restructuring Tool) has been created which utilises this part-completion method.

ED 477 013

Introduction

In education, certain subjects require problem solving skills and are considered by many students to be inherently difficult. An example of such a subject is software development and this requires students to be able to analyse problems and then design and implement solutions in a computer programming language. It has been observed that a large number of students achieve only low grades and become disillusioned with the subject, for example Perkins, Schwartz, & Simmons (1988) state that:

> *Students with a semester or more of instruction often display remarkable naivete about the language that they have been studying and often prove unable to manage dismayingly simple programming problems.*

Also, King (1994) states:

> *Even after two years of study, many students had only a rudimentary understanding of programming.*

One of the reasons for the above is that students experience a very high cognitive load during their learning and this paper proposes that cognitive load theory needs to be carefully taken into account in the design of learning materials and tools for such problem solving domains. The paper then describes a software tool called CORT (Code Restructuring Tool) that has been built by the author and used with novice programmers at a university in Australia.

Cognitive Load Theory

Cognitive load theory is built upon the idea that working memory is limited to around seven chunks of material (Miller, 1956) and that people can only deal with two or three elements simultaneously. The degree of interactivity between the elements also affects the capacity of working memory.

Chess playing can be considered a problem solving domain and research (Chase & Simon, 1973) showed that the main difference between novices and experts was the fact that the latter had thousands of board configurations, as many as 100000 (Simon & Gilmartin, 1973), stored in long-term memory within schemata. The consequence is that, unlike less-skilled players, experts do not have to spend as much time searching for good chess moves using their limited working memory. Similarly, research into problem solving (Carroll, 1994) confirmed that, compared to novices, experts have knowledge of an enormous number of problem states and their associated moves. Such states are within long-term memory and such research indicates that human problem solving comes from stored knowledge and not from complex reasoning within working memory. It is

IR021741

BEST COPY AVAILABLE

suggested that humans are poor at complex reasoning unless most of the elements with which we reason are already in long-term memory, working memory being incapable of highly complex interactions using novel elements (Sweller, van Merrienboer, & Paas, 1998). This means that novices who are attempting a problem must engage in complex chains of reasoning using their working memory and in doing so it is likely that working memory will be overburdened. In other words the cognitive load on novices is too great.

Ways in which cognitive load can be reduced for novice problem solvers are therefore very important. In the schema theory of model representation, a schema can be anything that can be treated as a single entity or element such as a mathematical formula or a particular programming algorithm. Schemata have the function of storing knowledge and reducing the burden on working memory.

Experts can process information relevant to their domain automatically, novices however having to process information consciously (Schneider & Shiffrin, 1977; Tindall-Ford, Chandler, & Sweller, 1996). An example of such automatic processing is that of the expert driver who can drive their car without apparently thinking, whereas a learner driver has to consciously think of several things at the same time such as depressing the clutch and shifting to a new gear, observing the road ahead, moving the steering wheel etc. Any instructional design for a domain has to therefore not only encourage the construction of sophisticated schemata but also encourage the automatic processing of those schemata. This is important because of the limited capacity of working memory that can only deal with a few schemata at the same time. The ease with which information can be processed in working memory is the main thrust of cognitive load theory.

Working memory may be affected by intrinsic cognitive load and extraneous cognitive load (Sweller, 1994). In recent research, a further distinction is made with the inclusion of germane cognitive load (Sweller et al., 1998).

Intrinsic Cognitive Load

Intrinsic cognitive load is determined by the mental demands of the task (Chandler & Sweller, 1996). Some material has very low cognitive load and an example is the learning of the basic vocabulary of a foreign language. Each element or schema is independent from the others with no interactivity and subsequently the required mental processing, or intrinsic cognitive load, is low. Tasks that have low element interactivity can be learnt serially rather than simultaneously. Tasks with a high degree of element interactivity have a heavy intrinsic cognitive load and an example is the learning of the grammar of a foreign language as all the words in phrases need to be considered, that is processed, at once.

Computer programming is a domain with a high intrinsic cognitive load and this needs to be recognised in any instructional design. The intrinsic cognitive load cannot be reduced, however something can be done about the extraneous cognitive load.

Extraneous Cognitive Load

Extraneous cognitive load is generated by the instructional format used in the teaching and learning process and poor design leads to a high extraneous cognitive load. If a high extraneous cognitive load is combined with a high intrinsic cognitive load then this can lead to working memory overload. This is often what happens with novice programmers when the instructional design is poor.

The important point is that when the intrinsic cognitive load of the material is high, then it is incumbent on the instructional designer to think very carefully and ensure that the extraneous cognitive load is as low as possible. A lot of research has been done in looking at ways of reducing extraneous cognitive load, for example (Chandler & Sweller, 1991; Kalyuga, Chandler, & Sweller, 1998; Marcus, Cooper, & Sweller, 1996; Sweller, 1994; Tindall-Ford et al., 1997). These include: integrating diagrams and text so as to reduce the "split-attention" effect; goal-free problem solving; and the use of worked examples in problem solving.

Germane Cognitive Load

More recently, the concept of germane cognitive load has been introduced into cognitive load theory (Sweller et al., 1998). It is thought that if the instructional design is such that the extraneous cognitive load is kept to a minimum, and the intrinsic cognitive load is not too high, then there may be some unused working

memory available. This could then be used by learners, with appropriate instructional design, to engage in conscious processing that helps in the construction of schemata in the particular domain of interest. This conscious processing is the germane cognitive load. An example is the use of part-complete solutions in the learning of problem solving (Paas, 1992; van Merrienboer, 1990; Van Merrienboer & De Croock, 1992). The studying of complete worked examples by students is seen as one way of reducing the extraneous cognitive load. When students have to complete an incomplete worked example then they have to "mindfully abstract" the schemata from the example in order to understand it. That is, they have to consciously process it and this increases the germane cognitive load. Figure 1 shows the relationship between the various cognitive loads in the domain of programming.

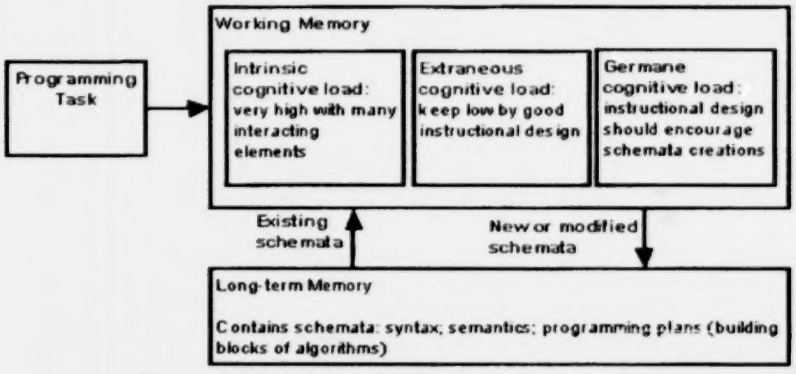
Working Memory
Programming Task
Intrinsic cognitive load: very high with many interacting elements
Extraneous cognitive load: keep low by good instructional design
Germane cognitive load: instructional design should encourage schemata creations
Existing schemata
New or modified schemata
Long-term Memory
Contains schemata: syntax; semantics; programming plans (building blocks of algorithms)

Figure 1: Cognitive load relationships in programming

Use of Worked Examples and Part-Complete Solutions in the Teaching and Learning of Programming

The "Reading" approach to the learning of programming makes use extensive of worked examples in an attempt to reduce the extraneous cognitive load on novices. A lot of the work in this area has been carried out by van Merrienboer and his colleagues (van Merrienboer, 1990a; van Merrienboer, 1990b; van Merrienboer & Paas, 1990c; van Merrienboer & De Croock, 1992; van Merrienboer, Krammer, & Maaswinkel, 1994). They argue that the traditional approach to the teaching and learning of programming is ineffective and that the "Reading" approach is a better one to follow. However, they also suggest that presenting worked examples to students is not sufficient as the students may not "abstract" the programming plans from them. Programming plans can be considered analogous to the chess of board configurations mentioned earlier. "Mindful" abstraction of plans is required by the voluntary investment of effort and the question then arises as to how we can get students to study the worked examples properly. In practice, students tend to rush through the examples, even if they have been asked to trace them in a debugger, as they often believe that they are only making progress in their learning when they are attempting to solve problems.

Such conscious processing by students places germane cognitive load upon them. One suggestion is that students should annotate worked examples with information about what they do or what they illustrate (Lieberman, 1986). Another suggestion is to use incomplete, well-structured and understandable program examples that require students to generate the missing code or "complete" the examples. This latter approach forces students to study the incomplete examples as it would not be possible for their completion without a thorough understanding of the examples' workings. An important aspect is that the incomplete examples are carefully designed as they have to contain enough "clues" in the code to guide the students in their completion. In other words, the germane cognitive load must not be made too large. It is suggested that this method facilitates both automation, students having blueprints available for mapping to new problem situations, and schemata acquisition as they are forced to mindfully abstract these from the incomplete programs (van Merrienboer & Paas, 1990).

BEST COPY AVAILABLE

In one study, two groups of 28 and 29 high-school students from grades 10 to 12 participated in a ten lesson programming course using a subset of COMAL-80 (van Merrienboer, 1990b). One group, the "generation" group, followed a conventional approach to the learning of programming that emphasised the design and coding of new programs. The other group, the "completion" group, followed an approach that emphasised the modification and extension of existing programs. It was found that the completion group was better than the generation group in constructing new programs. It was found that the percentage of correctly coded lines was greater and that looping structures were more often combined with correct variable initialisation before a loop together with the correct use of counters and accumulators within the loop. It would appear that the completion strategy had indeed resulted in superior schemata formation for those students within that group. In addition, the completion group used superior comments in connection with the scope and goals of the programs, indicating that they had developed better high-level templates or schemata. It was noted in the study however that both groups were equal in their ability to interpret programs and that this might indicate that students in the completion group do not understand their acquired templates.

A side effect of the research was also noted. The drop-out rate from the completion group was found to be lower than for the generation group, particularly for female students with low prior knowledge. It was suggested that perhaps the generation of complete programs is perceived as a difficult and menacing task and that the completion strategy overcomes this difficulty.

CORT (Code Restructuring Tool)

A tool has been designed by the author that is based upon the above ideas of the completion of part-complete programming solutions. Three methods of using CORT have been identified that allow various degrees of germane cognitive load to be applied and the tool has been utilised with students at Edith Cowan University in Australia, student feedback being extremely positive.

CORT has been used within a software development unit in a Business degree course. This unit is an introductory unit to programming in Visual BASIC, the majority of students having no previous programming experience. The unit runs over a period of 14 weeks, the students having a two-hour lecture and a one-hour computer laboratory session in which they attempted various "CORT" problems. The students would then finish their problems if necessary in their own time.

A Typical "CORT" Computer Laboratory Session

At the beginning of a computer laboratory session, the students are given a hard copy of a problem statement that they have to try and solve using CORT. The students would then run the CORT program and load in the CORT file corresponding to that particular problem. Two windows display a part-complete solution to the problem together with possible lines to be used as shown in figure 2.

The part-complete solution on the right is then completed by moving certain lines from the left hand window. Lines can also be moved up or down, and indented or outdented in the right hand window. Some problems may have too many lines in the left hand window, some of those lines being incorrect. In addition, there is a simple editor to allow the amendment, addition, and deletion of lines of code.

When the solution has been completed, the code from the right-hand window is copied to the Windows clipboard. Visual BASIC is then run and a file is loaded which contains no programming code but does contain the necessary Visual BASIC interface for the problem under consideration. The CORT code is then pasted into Visual BASIC and the program tested. Use is made of the trace and debugging facilities of Visual BASIC, these facilities providing an insight to the workings of the notional machine.

If the program does not work, a student can switch back to CORT and make any necessary changes before retesting that code in Visual BASIC.

Methods of using CORT

Three methods of using CORT were identified and these different methods were varied throughout the unit so that different levels of germane cognitive load could be applied. The first method is to supply all the missing lines of code in the left-hand window without any extra lines. The second method also supplies all the missing

lines of code but also includes extra "distracter" lines of code that are not required to complete the program solution. Finally the third method is to supply only some of the lines of code in the left-hand window, students having to key-in some lines themselves.

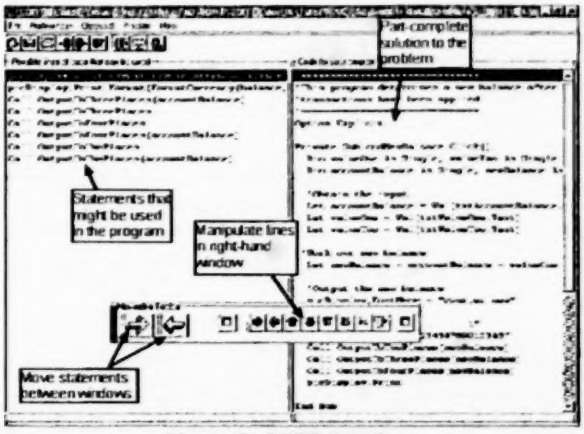
Part-complete solution to the problem
Statements that might be used in the program
Manipulate lines in right-hand window
Move statements between windows

Figure 2: CORT interface

Preliminary Feedback from Students

Data was collected on the use of CORT during semester 2, 2001. Preliminary results suggest the following.

- Students commented on the fact that CORT provides a starting point for solving a problem. Many suggested that they would not know how to go about tackling some of the problems if CORT was not provided.
- Method 2 of using CORT proved to be most popular. Method 1 was seen by students as being too easy while method 2 required students to think a great deal about the problem solution. Method 3 was not as popular as that required students to have knowledge of the exact syntax of statements.
- It was observed that students engaged in significant reflection and higher order thinking when using method 2.
- Students commented that they were encouraged and motivated by the fact that they could get their programs to work in a relatively short time frame.

Conclusions

This paper argues that cognitive load theory needs to be carefully taken into account when designing instructional materials for computer programming. Programming has a very high intrinsic cognitive load and therefore the extraneous cognitive load should be made as low as possible. However we still need to ensure that students "think" and are encouraged to create the necessary schemata in long term memory and this can be done by applying germane cognitive load upon them. One way of doing this is to carefully design solutions to problems that are only part complete and that require completion by students. This approach has been built into a new software tool called CORT (Code Restructuring Tool) (Garner, 2000) that reduces the extraneous cognitive load by supporting the completion approach to learning programming. Three methods of CORT have been identified and preliminary results suggest that method 2, in which all missing lines plus distracters are available in the left-hand window, has great potential for providing the necessary amount of germane cognitive load to help students develop the necessary programming schemata.

References

BEST COPY AVAILABLE

Carroll, W. (1994). Using worked examples as an instructional support in the algebra classroom. *Journal of Educational Computing Psychology, 86*, 360-367.

Chandler, P., & Sweller, J. (1991). Cognitive load theory and the format of instruction. *Cognition and Instruction, 8*, 293-332.

Chandler, P., & Sweller, J. (1996). Cognitive Load while Learning to use a computer program. *Applied Cognitive Psychology, 10*, 151-170.

Chase, W. G., & Simon, H. A. (1973). The Mind's Eye in Chess. In W. G. Chase (Ed.), *Visual Information Processing*. New York: Academic.

Garner, S. (2000, 21-24 Nov). *A Code Restructuring Tool to help Scaffold Novice Programmers*. Paper presented at the International Conference in Computer Education, ICCE2000, Taipei, Taiwan.

Kalyuga, S., Chandler, P., & Sweller, J. (1998). Levels of expertise and instructional design. *Human Factors, 40*, 1-17.

King, J., J. Feltham, et al. (1994). "Novice Programming in High Schools: Teacher Perceptions and New Directions." Australian Educational Computing(Sep 1994,): 17-23.

Lieberman, H. (1986). An Example Based Environment for beginning Programmers. *Instructional Science, 14*(3), 277-292.

Marcus, N., Cooper, M., & Sweller, J. (1996). Understand instructions. *Journal of Educational Psychology, 88*, 49-63.

Miller, G. A. (1956). The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity to Process Information. *Psychological Review*(63), 81-97.

Paas, F. G. W. C. (1992). Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive load approach. *Journal of Educational Psychology, 84*, 429-434.

Perkins, D. N., Schwartz, S., & Simmons, R. (1988). Instructional Strategies for the Problems of Novice Programmers. In R. E. Mayer (Ed.), *Teaching and Learning Computer Programming: Multiple Research Perspective*(pp. 153-178). Hillsdale, NJ: Erlbaum.

Schneider, W., & Shiffrin, R. (1977). Controlled and automatic human information processing: Detection, search and attention. *Psychological Review, 84*, 1-66.

Simon, H., & Gilmartin, K. (1973). A Simulation of memory for Chess Positions. *Cognitive Psychology, 5*, 29-46.
Sweller, J. (1994). Cognitive load theory, learning difficulty, and instructional design. *Learning and instruction, 4*, 295-312.
Sweller, J., van Merrienboer, J. J. G., & Paas, F. G. W. C. (1998). Cognitive Architecture and Instructional Design. *Educational psychologyreview, 10*(3 SEP 01 1998), 251-.

Tindall-Ford, S., Chandler, P., & Sweller, J. (1997). When two sensory modes are better than one. *Journal of Experimental Psychology: Applied, 3*, 257-287.

van Merrienboer, J. J. G. (1990a). Instructional Strategies for Teaching Computer Programming: Interactions with the Cognitive Style Reflection-Impulsivity. *Journal of research on computing in education, 23 Fall 1990*(1), 45-.

van Merrienboer, J. J. G. (1990b). Strategies for Programming Instruction in High School: Program Completion vs. Program Generation. *Journal of educational computing research., 6*(3), 265-.

van Merrienboer, J. J. G., & Paas, F. (1990c). Automation and Schema Acquisition in learning elementary computer programming. *Computers in Human Behavior*(6), 273-289.

van Merrienboer, J. J. G., & De Croock, M. B. M. (1992). Strategies for computer-based programming instruction: program completion vs. program generation. *Journal of Educational Computing Research, 8*(3), 365-394.

van Merrienboer, J. J. G., Krammer, H. P. M., & Maaswinkel, R. M. (1994). Automating the planning and construction of programming assignments for teaching introductory computer programming. In R. D. Tennyson (Ed.), *Automating Instructional Design, Development, and Delivery (NATO ASI Series F, Vol. 119)* (pp. 61-77). Springer Verlag, Berlin.

BEST COPY AVAILABLE

U.S. Department of Education
Office of Educational Research and Improvement (OERI)
National Library of Education (NLE)
Educational Resources Information Center (ERIC)

NOTICE

Reproduction Basis

[X] This document is covered by a signed "Reproduction Release (Blanket)" form (on file within the ERIC system), encompassing all or classes of documents from its source organization and, therefore, does not require a "Specific Document" Release form.

[ ] This document is Federally-funded, or carries its own permission to reproduce, or is otherwise in the public domain and, therefore, may be reproduced by ERIC without a signed Reproduction Release form (either "Specific Document" or "Blanket").